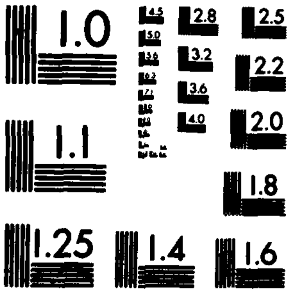


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

PHOTOGRAPH THIS SHEET

AD A110004

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Intermetrics, Inc.
Cambridge, MA
ADA Integrated Environment I Computer Program Development Specification. Interim Rept.
15 Sep. 80 - 15 Mar 81
Dec. 81

DOCUMENT IDENTIFICATION

Contract F30602-80-C-0291 RADC-TR-81-358, Vol. VI

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR
NTIS GRA&I ☒
DTIC TAB □
UNANNOUNCED □
JUSTIFICATION

BY
DISTRIBUTION /
AVAILABILITY CODES

DIST	AVAIL AND/OR SPECIAL	
A		Code 23

DISTRIBUTION STAMP

DATE ACCESSIONED

82 01 12 013

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

RADC-TR-81-358, Vol VI (of seven)
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

Intermetrics, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This document was produced under Contract F30602-80-C-0291 for the Rome Air Development Center. Mr. Don Roberts is the COTR for the Air Force. Dr. Fred H. Martin is Project Manager for Intermetrics.

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-358, Volume VI (of seven) has been reviewed and is approved for publication.

APPROVED:

DONALD F. ROBERTS
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
RADC-TR-81-358, Vol VI (of seven)

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

5. TYPE OF REPORT & PERIOD COVERED
Interim Report
15 Sep 80 - 15 Mar 81

6. PERFORMING ORG. REPORT NUMBER
N/A

7. AUTHOR(s)

8. CONTRACT OR GRANT NUMBER(s)
F30602-80-C-0291

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Intermetrics, Inc.
733 Concord Avenue
Cambridge MA 02138

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
62204F/33126F
55811908

11. CONTROLLING OFFICE NAME AND ADDRESS
Rome Air Development Center (COES)
Griffiss AFB NY 13441

12. REPORT DATE
December 1981

13. NUMBER OF PAGES
33

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
Same

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
Same

18. SUPPLEMENTARY NOTES
RADC Project Engineer: Donald F. Roberts (COES)

Subcontractor is Massachusetts Computer Assoc.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This B-5 Specification describes, in detail, the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

APSE is built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this specification include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

TABLE OF CONTENTS

PAGE

1.0 SCOPE 1

1.1 Identification 1
1.2 Functional Summary 1

2.0 APPLICABLE DOCUMENTS 3

2.1 Government Documents 3
2.2 Non-Government Documents 3

3.0 REQUIREMENTS 5

3.1 Program Definition 5
3.2 Detailed Functional Requirements 6

3.2.1 Parameters 6
3.2.2 Inputs 7

3.2.2.1 General Information 7
3.2.2.2 Breakpoint Commands 12
3.2.2.3 Execution Control Commands 15
3.2.2.4 Information Commands 16
3.2.2.5 Debug Control Commands 18

3.2.3 Outputs 19
3.2.4 Processing 19

3.2.4.1 Command Processing/MAPSE Command Processor Interface 19
3.2.4.2 Debug/Compiler Interface 19
3.2.4.3 Debug/RTS-KAPSE Interface 21
3.2.4.4 Debug/Internal Data Base Interface 24
3.2.4.5 Debug/Program Library Access Package 24
3.2.4.6 Breakpoint Command Procedures 25
3.2.4.7 Execution Control Command Procedures 27
3.2.4.8 Information Command Procedures 29
3.2.4.9 Debug Control Command Procedures 31
3.2.4.10 Utility Procedures 31

TABLE OF CONTENTS (Cont'd.)

PAGE

3.3 Adaptation 32

3.3.1 Debug Size Restrictions 32
3.3.2 Debug Extensions 32
3.3.3 Run-Time System Parameters 32

4.0 QUALITY ASSURANCE PROVISIONS 33

LIST OF FIGURES

FIGURE 3-1: Sample Debugging Session 8
FIGURE 3-2: Debug Procedures and Interfaces 20

1.0 SCOPE

This specification describes the MAPSE debugging facilities. It includes descriptions of: (1) the user debugging language; (2) the DEBUG computer program that serves as an interface between the user and these facilities; and (3) the interface between DEBUG and other MAPSE components, through which debugging tasks are performed.

1.1 Identification

The MAPSE debugging facilities support full source level debugging of Ada programs. These facilities draw upon several components of other MAPSE tools. Use of these components for debugging purposes is coordinated by the MAPSE tool, Debug. Debug accepts and interprets user input commands and causes the specified actions to be performed, calling upon other tools, as needed.

This document contains specifications for the Debug command syntax and semantics; specifications describing how the Debug commands are effected; and specifications of the debugging support functions provided by other MAPSE components. The MAPSE components that provide explicit support for the MAPSE debugging facilities include:

1. an interactive debugger (Debug) that allows the user to manipulate an executing program;
2. the Ada Compiler;
3. the MAPSE Linkage Editor;
4. the Ada Program Run Time System (RTS);
5. the KAPSE, for supporting the Debug interface with the RTS; and
6. The MAPSE Command Processor (MCP).

Tools other than Debug are described in detail in separate B-5 specifications, although some of their requirements are specified here.

1.2 Functional Summary

Debug itself is a command language interpreter. It accepts, as input, commands from the user, interprets them, and then performs the actions requested. Most often, user commands establish information in a small data base maintained by Degug, display information about the executing user program, or control its execution. The user can, among other things, display trace information, display and modify the values of variables in the user program, change the execution flow in the program, and establish breakpoints at which program execution can be interrupted. Commands can be issued interactively or can be stored in text files that serve as command scripts to Debug.

The Debug support functions within MAPSE components provide Degug with information about the program being debugged and provide Debug with the basic elements of execution control over the program. All of these other MAPSE components provide their portion of the Debug support automatically, so that the Debug tool itself is the primary user interface with the MAPSE debugging facilities.

2.0 APPLICABLE DOCUMENTS

Please note that the bracketed number preceding the document identification is used for reference purposes within the text of this document.

2.1 Government Documents

2.2 Non-Government Documents

[I-1] System Specification for Ada Integrated Environment, Type A, Intermetrics, Inc., March 1981, IR-676.

Computer Program Development Specifications for Ada Integrated Environment (Type B5):

[I-2] Ada Compiler Phases, IR-677

[I-3] KAPSE/Database, IR-678

[I-4] MAPSE Command Processor, IR-679

[I-5] MAPSE Generation and Support, IR-680

[I-6] Program Integration Facilities, IR-681

[I-7] MAPSE Text Editor, IR-683

[I-8] Technical Report (Interim), IR-684

3.0 REQUIREMENTS

Debug is an interactive MAPSE tool that allows a user to control an executing program and examine its state. To do this, Debug requires support from a variety of MAPSE components. These requirements are fully specified in later sections of this document.

Debug requires no peripheral equipment other than the terminal or file that represents an input script. In addition, a terminal or set of files are required to record the debugger output.

The peripheral programs required by Debug are only summarized here. They are handled in detail in Section 3.2.

Debug needs to identify the user program being debugged. This is specified by the user when he invokes Debug from the MCP or from some other tool. See Section 3.2.1.

Debug needs access to the Program Library for the program being debugged. This library contains information required by Debug, such as symbol-definition information, addresses of program entities, and cross listings. The information in this library is created by the Ada Compiler and the Link Editor. See sections 3.2.4.2 (Debug/Compiler Interface) and 3.2.4.5 (Debug/Program Library Access).

Debug requirements of the Ada Run Time System and KAPSE are described in section 3.2.4.3 (Debug RTS/KAPSE Interface).

3.1 Program Definition

The Debug program is composed of several different components, as summarized below.

There is a Debug command processor; it is derived from the MAPSE Command Processor, with the addition of specific verbs to implement specific debugging functions.

There is a set of procedures that implement the specific commands, generally one procedure per command.

There is also a set of utility procedures to parse Ada names with subscripting and qualification, analyze them in the context of the user program, and evaluate expressions interpretively.

There are Ada packages that handle the interfaces between Debug and the various other tools with which it interacts. These include the Debug Support Routine in the Ada Run-Time System and the program library access package.

Although Debug does not have a direct executable interface with the Compiler, it does require Compiler support. This functionality is described in Section 3.2.4.2 (Debug/Compiler Interface).

To maintain information concerning which breakpoints are active and to keep stored user commands, Debug has an internal data base. This is implemented as an abstract data type via an Ada package specification.

3.2 Detailed Functional Requirements

Section 3.2.1 describes the set of parameters that must be supplied to Debug when it is invoked.

Section 3.2.2 describes the additional commands provided by the Debug Command Processor and shows, by examples, how to use the Debug commands within the MAPSE Command Processor to perform tasks such as installation of conditional breakpoints.

Section 3.2.3 summarizes the outputs of Debug.

Section 3.2.4 describes the various modules and interfaces that Debug uses to effect the Debug-specific commands. These include the Debug Command Processor itself, the execution procedures it calls to do the actual work, the interface with the Ada Compiler, the interface with the Ada Run-Time System through the KAPSE, the interface with the Program Library Access Package, and Debug's own internal data base. The interfaces are specified in terms of requirements that Debug imposes upon the other MAPSE components in order to provide the specified funtionality.

3.2.1 Parameters

[I-4] describes how the user can invoke a MAPSE tool from the MCP. Debug is a MAPSE tool, invoked with four parameters. The first two, respectively, specify the source from which Debug commands are input, and the source to which output is to be directed. Normally, these are the standard input and standard output devices, namely the user's terminal. In the case that Debug is being run in the background, these are a script of Debug commands and a file to write output to; the MCP interface handles I/O direction.

The third parameter to Debug specifies which program the debugger is to control. This argument is a window on the context of the user program. See [I-3] for a description of "window" and "context". The fourth parameter is an ASCII string containing the parameters to be passed to the user program being debugged when it started. If the user program has already started when Debug is called, this fourth parameter is ignored.

3.2.2 Inputs

The Debug command language includes as a subset the entire MCP command language. The extension is in the form of Debug-specific commands which generally have a verb-object structure. The following subsections present a brief description of the form and function of each Debug-specific command. Later sections provide a more detailed discussion of the effect of each command and the specification of how it will be implemented.

Since the Debug command language is an extension of the MCP command language, the following subsections distinguish between MCP functions and Debug-specific commands. Any to MCP variables or commands, in fact, refer to the MCP functions contained within the Debug Command Processor. Figure 3-1 is a sample debugging session.

3.2.2.1 General Information

(a) Breakpoint Locations. A breakpoint represents a place in a program where execution has been suspended to allow the user to examine various aspects of the program. Usually, this place is before a statement or elaboration item. (In the case of highly optimized modules this may not be the case, see Section 3.2.4.2). For the purposes of the debugging environment, each declaration in the user program is also considered a statement. This means that the user can establish breakpoints between declarations. This reflects the Ada language rule concerning the order of elaboration of declarations.

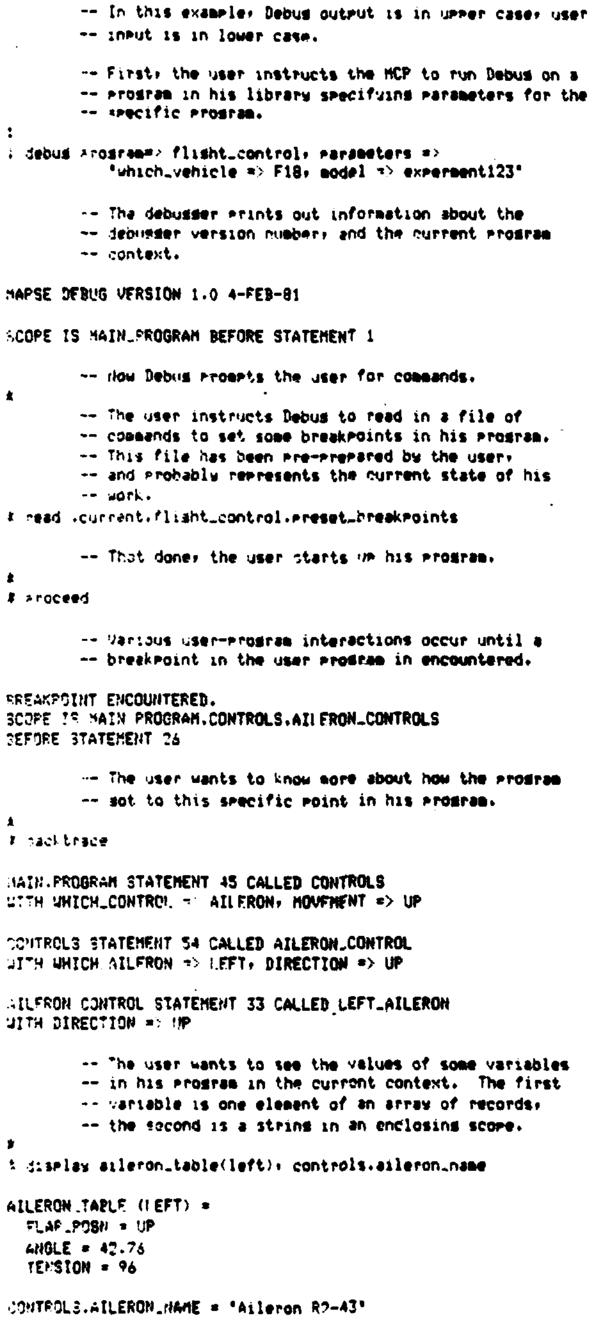
User commands to Debug can be seen as executing at the specific point that the user program is halted. Thus, alterations to the flow of the user program by Debug commands occur before executing the next statement or elaboration item after the breakpoint.

(b) Scopes. Each statement or elaboration item is contained within some scope in an Ada program, and Debug preserves this viewpoint for the user. This means that all variables visible in the current scope where the program is halted are also visible to the user. (At a breakpoint in the middle of a declarations section, only some of the declarations have been elaborated, so the user is restricted to by Ada elaboration order rules when issuing Debug commands.) Other variables in other scopes are available via normal Ada name qualification. Debug also provides commands to alter the scope that the user sees while at a breakpoint.

(c) Command Overview. There are four classes of user commands: (1) breakpoint commands; (2) execution control commands; (3) information commands; and (4) Debug control commands.

Breakpoint commands allow the user to set, remove, suspend and restore various breakpoints and breakpoint actions.

Execution control commands allow the user to stop, start, and modify the execution of his program.



```
          -- In this example, Debug output is in upper case, user
          -- input is in lower case.

          -- First, the user instructs the MCP to run Debug on a
          -- program in his library specifying parameters for the
          -- specific program.
:
: debug program=> flight_control, parameters =>
          'which_vehicle => F18, model => experment123'

          -- The debugger prints out information about the
          -- debugger version number, and the current program
          -- context.

MAPSE DEBUG VERSION 1.0 4-FEB-81

SCOPE IS MAIN_PROGRAM BEFORE STATEMENT 1

          -- Now Debug prompts the user for commands.
*
          -- The user instructs Debug to read in a file of
          -- commands to set some breakpoints in his program.
          -- This file has been pre-prepared by the user,
          -- and probably represents the current state of his
          -- work.
* read .current.flight_control.preset_breakpoints

          -- That done, the user starts up his program.
*
* proceed

          -- Various user-program interactions occur until a
          -- breakpoint in the user program in encountered.

BREAKPOINT ENCOUNTERED.
SCOPE IS MAIN PROGRAM.CONTROLS.AILERON_CONTROLS
BEFORE STATEMENT 26

          -- The user wants to know more about how the program
          -- got to this specific point in his program.
*
* backtrace

MAIN.PROGRAM STATEMENT 45 CALLED CONTROLS
WITH WHICH_CONTROL => AILERON, MOVEMENT => UP

CONTROLS STATEMENT 54 CALLED AILERON_CONTROL
WITH WHICH AILERON => LEFT, DIRECTION => UP

AILERON CONTROL STATEMENT 33 CALLED LEFT_AILERON
WITH DIRECTION => UP

          -- The user wants to see the values of some variables
          -- in his program in the current context.  The first
          -- variable is one element of an array of records,
          -- the second is a string in an enclosing scope.
*
* display aileron_table(left), controls.aileron_name

AILERON_TABLE (LEFT) =
  FLAP_POSN = UP
  ANGLE = 42.76
  TENSION = 96

CONTROLS.AILERON_NAME = 'Aileron R2-43'
```

FIGURE 3-1: Sample Debugging Session

```
        -- The user sets a breakpoint with stored commands at
        -- every location that the variable 'master_switch'
        -- might be modified, and then proceeds.
*
* trap modify aileron control.master_switch begin
  1/  display master_switch
  2/  set master_switch := off
  3/  proceed
  4/  end
* proceed

MASTER_SWITCH = ON
MASTER_SWITCH = OFF
MASTER_SWITCH = ON
   ...

BREAKPOINT ENCOUNTERED.
SCOPE IS MAIN.PROGRAM.CONTROLS.AILERON_CONTROLS
BEFORE STATEMENT 26

        -- The user realizes he wants a different set
        -- of commands at the same breakpoint.  He asks
        -- for a list of breakpoints to check that he has
        -- all that he wants, then removes this latest
        -- breakpoint by using the breakpoint identifier,
        -- and inserts a new one.  He then instructs
        -- Debug to transfer to a program label.
*
* what trap

   ...
 <<CONTROL_POINT>> TRAP BEFORE STMT
         MAIN.PROGRAM.CONTROLS.AILERON_CONTROLS.26
 <<BKPT_21>> TRAP MODIFY AILERON_CONTROL.MASTER SWITCH BEGIN
   1/  DISPLAY MASTER_SWITCH
   2/  SET MASTER_SWITCH := OFF
   3/  PROCEED
   4/  END

* remove bkpt_21
* modify aileron_control.master_switch begin
   1/  what scope
   2/  display master_switch
   3/  set master_switch := off
   4/  proceed
   5/  end
* goto label calculate_aileron

SCOPE IS MAIN.PROGRAM.CONTROLS.WING_TIP_LEFT
BEFORE STATEMENT 16
MASTER_SWITCH = ON
SCOPE IS MAIN_PROGRAM.CONTROLS.AILERON.CONTROLS
BEFORE STATEMENT 14
MASTER_SWITCH = ON
SCOPE IS MAIN_PROGRAM.CONTROLS.AILERON_CONTROLS.MOVE_UP
BEFORE STATEMENT 66
MASTER_SWITCH = OFF
  ...

        -- The user is closing up for the day, and saves
        -- his current breakpoint set, and quits.
*
* save preset_breakpoints
* quit
: logout
```

FIGURE 3-1: Sample Debugging Session (Con't.)

Information commands allow the user to inspect the state of his program by examining the values of variables, the current location of execution and the history of execution (call chain, task scheduling, rendezvous, jump flow).

Debug control commands are used to leave the debugger, read in command files, and redirect output.

(d) Language Overview. The user may specify any legal Ada variable in normal Ada syntax for Debug DISPLAY and SET commands. This includes access dereferencing, array subscripting and record component selection. Ambiguous names may be qualified using normal Ada syntax as well. The expressions used as array subscripts may not include function calls. In addition, operators are not resolved to any overloaded definition.

All lists of identifiers are separated by commas. These include variable names, statement identifiers and exception names.

Debug commands are terminated by semicolon or newline. The stored command part of breakpoint commands is terminated only by the matching "end".

Statement identifiers and scope identifiers are specified using a simple extension to Ada name qualification. Statements are referred to by suffixing their procedure name by a dot followed by the sequential statement number relative to the start of the procedure. This is the same statement number used by the Ada Compiler listing option. To avoid using statement numbers in Debug commands, a user can put Ada labels on those statements and use the LABEL option of the TRAP and GOTO Debug commands.

A procedure is referred to by name, unless it is overloaded. In that case, it is referred to by its qualified name, followed by a number sign (#) and the sequential specification number of the desired procedure. This number appears in the compiler listing. That is, given five procedures P in the current scope, P#3 refers to the third specification for P. This number appears in the program listing. For the purposes of the debugger, procedure specifications count as definitions of procedures, so if the program contains a specification for P and then a body for P, P#2 must be specified when referring to the body of P. In the case that this form is used to refer to a statement within an overloaded procedure, the specification used must be the procedure body specification, and not its interface or stub specification. This results in a name followed by 'number sign' <integer> 'dot' <integer>. When the user wants to put a breakpoint at a statement in the current scope, no procedure name qualifier is necessary.

Examples:

```
    BREAK BEFORE 5              -- sets breakpoint before statement 5
                                -- of current scope

    BREAK BEFORE proc^4.10      -- set trap before tenth statement of
                                -- fourth definition of proc in
                                -- the current scope

    BREAK before prog.25        -- trap before twenty-fifth statement
                                -- in scope "prog"

    SCOPE TO prog.proc          -- change scope to inside
                                -- procedure "proc" inside scope
                                -- "prog"
```

(e) Expressions and Variables. The MAPSE Command Language allows variables in all expressions. These variables are always MCP variables, and are prefixed by a percent sign (%). The Debug Command Processor recognizes user program variables in only a few contexts. In general, all expressions and variables are MCP expressions and variables. Only the SET, TRAP MODIFY, PGM_RETURN and DISPLAY commands recognize user variables. That is, the variables in the expressions in these statements must all be variables in the user program. The only exception is the destination variable of the SET command. It can be an MCP variable. (See the SET command, Section 3.2.2.3.) This provides the capability of writing Debug scripts that test the values of expressions and variables in the user program.

(f) Notation Conventions. The following command descriptions use the notation conventions: square brackets [] surround options; parentheses () surround optional words that help convey the meaning of the command; angle brackets <> surround higher-level constructs

3.2.2.2 Breakpoint Commands

Breakpoint commands establish breakpoints in the user program. No other action is taken.

Following is a summary of the breakpoint commands. Each keyword given below may be prefixed by the word TRAP or BREAK.

```
AFTER (STMT) <list of statement identifiers>

                                -- break after the specified
                                -- statements

BEFORE (STMT) <list of statement identifiers>

                                -- break before the specified
                                -- statements
LABEL <list of label identifiers>

                                -- break at specified labels

EVERY <number of statements>

                                -- break after every "n"
                                -- statements

MODIFY (OF) <list of variables>

                                -- break after each statement
                                -- that modifies the specified
                                -- variables

EXCEPTIONS <list of exceptions>

                                -- break on raise of these
                                -- exceptions

ALL_EXCEPTIONS                  -- break on raise of all
                                -- exceptions

UNHANDLED_EXCEPTIONS            -- break only on unhandled
                                -- exceptions

INVOKE (SUBPROGRAM)             -- break on entry to all
                                -- subprograms and entries

EXIT (SUBPROGRAM)               -- break on exit from all
                                -- subprograms and entries
```

Following are the breakpoint modification commands.

```
     DEACTIVATE <list of breakpoint identifiers>
                                -- suspends action of breakpoint

     REACTIVATE <list of breakpoint identifiers>
                                -- restores action of breakpoint

     REMOVE <list  of breakpoint  identifiers>
                                -- removes and  forgets the
                                -- specified breakpoints.
```

Commands that set breakpoints may be prefixed by an identifier within the Ada << and >> markers. This syntax indicates that the breakpoint can be referred to by the specific name enclosed within the brackets. The name serves as an abbreviation for all breakpoints specified by the command. The identifier must be a regular Ada identifier.

All breakpoints must have names; those not named by the user are given a unique name by Debug. If the user specifies a breakpoint command with a breakpoint identifier already in use, the name on the new breakpoint command is replaced by a Debug-generated name, and a warning message issued. Debug-generated breakpoint names are constructed by concatenating the string form of an integer with the letters "bkpt_". (Examples: bkpt_1, bkpt_465.)

Breakpoint commands may contain a sequence of Debug commands. These commands are executed when the breakpoint is encountered in the flow of the user program. After the commands have been executed, the user program execution is resumed at the point at which it was interrupted. If no sequence of commands is specified, the program execution is halted and the user is allowed to give Debug commands interactively. The PROCEED command may be one of the stored commands. In this case, the program will not be suspended for interactive debugging at the breakpoint.

These stored commands are specified by the keyword BEGIN following the breakpoint command on the same line. The stored commands may then be typed, one per line or separated by semicolons as in the MAPSE Command Language. The BEGIN is terminated by the matching occurrence of the keyword END. A stored command sequence can contain nested BEGIN-END blocks.

Breakpoints can be removed by use of the REMOVE command. The breakpoint and its stored actions can be suspended by use of the DEACTIVATE command. The REACTIVATE command restores the actions of a breakpoint that has been DEACTIVATEd. These three commands can be followed by a breakpoint identifier list. Thus entire groups of identifiers can be manipulated. The keyword ALL is permitted as a breakpoint identifier and specifies all breakpoints in the user program.

A second form of the REMOVE, DEACTIVATE and REACTIVATE commands is supplied to permit finer control over breakpoint insertion and deletion. Each of the breakpoint commands may be preceded by any of the three keywords of the breakpoint modification commands. For example:

```
REMOVE TRAP BEFORE CALC_SINE.24

DEACTIVATE MODIFY MASTER_SWITCH
```

Only the specified breakpoints are affected. These breakpoint commands following the modification keyword may not include the list of stored commands. (No BEGIN keyword is permitted at the end of the breakpoint modification command.)

The stored commands for a breakpoint provide the Debug user with conditional breakpoints. For example:

```
*
* TRAP BEFORE STMT 26 BEGIN

  1/ SET %M_S := MASTER_SWITCH

  2/ IF %M_S = ON THEN

  3/       PROCEED

  4/ ENDIF
```

This breakpoint command sets a breakpoint at statement 26 of the current scope. When that statement is encountered, the value of the program variable MASTER_SWITCH is fetched and stored in the MCP variable %M_S. The MCP variable is tested and, if found equal to the string "ON", causes the breakpoint to return control to the executing user program. Any other value of %M_S causes control to be automatically returned to the debug command processor. See the next section for an explanation of the SET and PROCEED commands.

Breakpoints installed in task bodies cause the breakpoint to occur at the specified point for all instances of the task. This implies that the breakpoint occurs in the next task instance that encounters the breakpoint. To have a breakpoint occur for a specific instance of a task, the Debug use can user the conditional breakpoint commands and SET command to fetch information from the program and check for the correct instanceof the task.

3.2.2.3 Execution Control Commands

Following is a list of Execution Control Commands.

```
SET <variable> := <expression>      -- modify value of
                                    -- specified variable

PGM_GOTO STMT <statement identifier>

                                    -- transfer of control

PGM_GOTO LABEL  <label identifier>
                                    -- transfer of control

PGM_RETURN   [ <expression>  ]
                                    -- return from subprogram

RAISE  [ <exception identifier> ]   -- raise exception

PROCEED [ <integer> ]               -- return control to
                                    -- executing program

STEP  [ <number of statements> ]    -- go "n" statements one
                                    -- at a time
```

Each command has an immediate effect. All but the SET command cause control to immediately return to the user program in the manner specified, thus closing the breakpoint. Control only returns to the Debug Command Processor when another breakpoint in the user program is encountered, or when the user hits the "interrupt" key.

The expression in the SET command follows the same rules as Ada expressions except that no user subprograms may be invoked. The implication of this is that overloaded operators may not be used, and no explicit function calls are allowed. In the case that the variable specified after the set keyword is an MCP variable (with prefix %), then the value is converted to a string and stored in the MCP variable. Only expressions of simple type (string, Boolean, integer, floating point, enumeral, access) are permitted when their value is being assigned to an MCP variable. In the case that the target of the assignment is a variable in a user program, then the effect is the same as an assignment statement in the Ada program. Debug permits the user to do assignment regardless of restrictions specified by the Ada language and enforced by the Compiler. This means that the user may assign objects of limited private type within a debugging session. The only rule is that the types must match.

The integer argument to the PROCEED command is optional and specifies the number of times to suspend the current breakpoint. Control is not returned to the Debug Command Processor from the current breakpoint until control has passed it the specified number of times. An argument of one (1) means skip the breakpoint once, and halt on the second occurrence. Thus the default for PROCEED is zero (0).

The RAISE command takes an optional argument, like the Ada language statement. It is the exception to be raised. When no exception is specified, the raise handler re-raises the current exception. The syntax for the exception name is the same as in the Ada LRM. This means that the user can raise the FAILURE exception in any task as well.

The number of statements to the STEP command is optional, and is defaulted to one.

The PGM_GOTO LABEL command has the same effect as the Ada goto statement; it causes program control to be transferred to the specified label. The PGM_GOTO STMT command allows transfer to any statement by statement identifier. The only checks on the PGM_GOTO commands are that the label or statement is in the subprogram or (package) enclosing the current breakpoint. The user is permitted to specify a GOTO that is illegal by Ada language rules.

The PGM_RETURN command has the same effect as the Ada return statement; it causes the subprogram enclosing the current breakpoint to terminate. If the subprogram is a function, the PGM_RETURN command must be followed by an expression which becomes the return value of the function.

The PGM_GOTO and PGM_RETURN commands are prefixed by "PGM_" to distinguish them from commands.

3.2.2.4 Information Commands

Following is a list of Information Commands.

```
DISPLAY <list of variables>  [ (IN) BASE <integer> ]

                              -- display value of variables
                              -- [in some base]

BACKTRACE [ CHAIN ]           -- show nested chain of procedure
                              -- calls of present breakpoint

BACKTRACE FLOW                -- show recent flow of control
                              -- of transfers (goto, if, etc)


BACKTRACE TASK                -- show recent task schedule and
                              -- rendevous history
```

```
BACKTRACE ALL                 -- do all backtrace options

SCOPE CALLER                  -- change scope to caller of
                              -- current scope

SCOPE ENCLOSING               -- change scope to static
                              -- enclosing scope

SCOPE RESET                   -- reset scope to original
                              -- breakpoint scope

SCOPE TO <scope identifier> -- change to arbitrary new scope

WHAT SCOPE                    -- print out current scope
                              -- (affected by "scope" cmd)

WHAT TRAP                     -- print list of current
                              -- breakpoints and status of
                              -- each breapoint

WHAT BREAK                    -- same as "what trap"

DUMP [ ALL ]                  -- print all variables in
                              -- current scope
```

The base option of the DISPLAY command is an integer from two (2) through sixteen (16) specifying the base of the numeric type desired and applies to the display of all variables in the variable list. When no base is supplied, the variable is printed out in its own mode (string, integer, floating point, enumeral, etc). When the variable is a composite object (array, record, etc) each component is printed in its own mode, unless a base was specified. In that case, all of the components of the variable are printed in the specified base.

The CHAIN keyword is the default for the BACKTRACE command so does not need to be specified.

The SCOPE commands only change the visibility of identifiers for the Debug commands. There is no affect upon the user program.

The DUMP command is a shorthand for displaying all the variables in the current scope. The optional keyword ALL indicates that all scopes should be dumped. Following the DUMP command by the MCP redirection of I/O operator and a file name (ex: DUMP ALL -> prog.dump) will place this information on the specified file rather than the user's terminal.

3.2.2.5 Debug Control Commands

Following are the Debug Control Commands. They are directives that control the execution of Debug rather than the user program.

```
SAVE (ON) <file>    -- save current breakpoints on text file

BRIEF               -- brief output to terminal only

VERBOSE             -- reverse effect of "brief" command

APPEND (TO) <file> [ ONLY ]

                    -- A copy of breakpoint and display
                    -- information is appended to the given
                    -- file.  No output to terminal when
                    -- ONLY specified.

NO APPEND           -- stop outputting to file

<interrupt>         -- stop the user program from executing
                    -- and return control to Debug
```

The BRIEF and VERBOSE commands shorten the prompt given the user when the executing program reaches a breakpoint.

The APPEND command differs from the redirection of I/O facility in that it affects all output actions of the debugger, not just a single command. The rationale is that a user will have a number of breakpoints displaying trace information that he now wants output to a file for the next piece of execution of his program. The output is appended to the specified file, so that the user can create an execution history on one file, rather than having to concatenate various files after debugging. If the file did not previously exist, it is created. The ONLY option of the output command causes no output to the user's terminal other than a brief report when a breakpoint is reached.

The user terminates a Debug session by using the RETURN command [I-4].

The SAVE command outputs a text file containing the necessary breakpoint commands to recreate the current state of Debug's data base. Only the breakpoints are recorded on this file, not current execution state, scope, etc. The MCP portion of the Debug Command Processor provides a general command file execution facility. This is used to read in the file containing the SAVEd breakpoints.

Example:

```
* SAVE PRESET_BKPTS

....

* EXEC CONTENTS(PRESET_BKPTS)
```

See [I-4] for explanation of the EXEC CONTENTS facility.

3.2.3 Outputs

Debug normally prompts the user with the character sequence asterisk, space (*) to indicate that it is accepting commands. The various Information Commands produce output according to their specifications. Whenever control returns to Debug from the user program, a report is issued summarizing the state of the user program. The BRIEF command makes this summary shorter. When the MCP I/O re-direction facilities are used, then various files are created. Files are also created by the Debug Control Command, APPEND.

3.2.4 Processing

The following subsections describe the processing of each major component, and provide a detailed description of the support provided by other MAPSE components. Figure 3-2 is an overview of Debug procedures and interface.

3.2.4.1. Command Processing/MAPSE Command Processor Interface

The Debug Command Processor (DCP) is created by taking a copy of the MAPSE Command Processor sources and extending them with a set of additional commands and command execution procedures. The changes to the MCP are all extensions, so that modifications to the MCP can easily be reflected in the DCP. This means that the Debug user sees the exact same functionality and syntax as when using the MAPSE Command Processor, with extensions for debugging activities.

One extension to the MCP is to enclose the command parser and interpreter procedures in a command_control procedure. This procedure is the top level control procedure of the DCP, and is responsible for checking the Debug data base whenever control is returned to it. Command_control checks to see if there are any pending commands (stored commands for the specific breakpoint or EXEC command). If so it executes them; otherwise it invokes the command parser and interpreter pair.

3.2.4.2 Debug/Compiler Interface

The Ada compiler has two user-specified parameters (not pragmas) that affect the functionality seen by the Debug user. These are DEBUG and OPTIMIZE.

The DEBUG parameter value of ON causes the compiler to insert "hooks" between every statement in the object code of each module compiled with this directive. See the next section (3.2.4.3 Debug/RTS-KAPSE Interface) for a discussion of hooks. Modules compiled without hooks (DEBUG parameter value of OFF) do not have the support necessary for use of statement breakpoints.

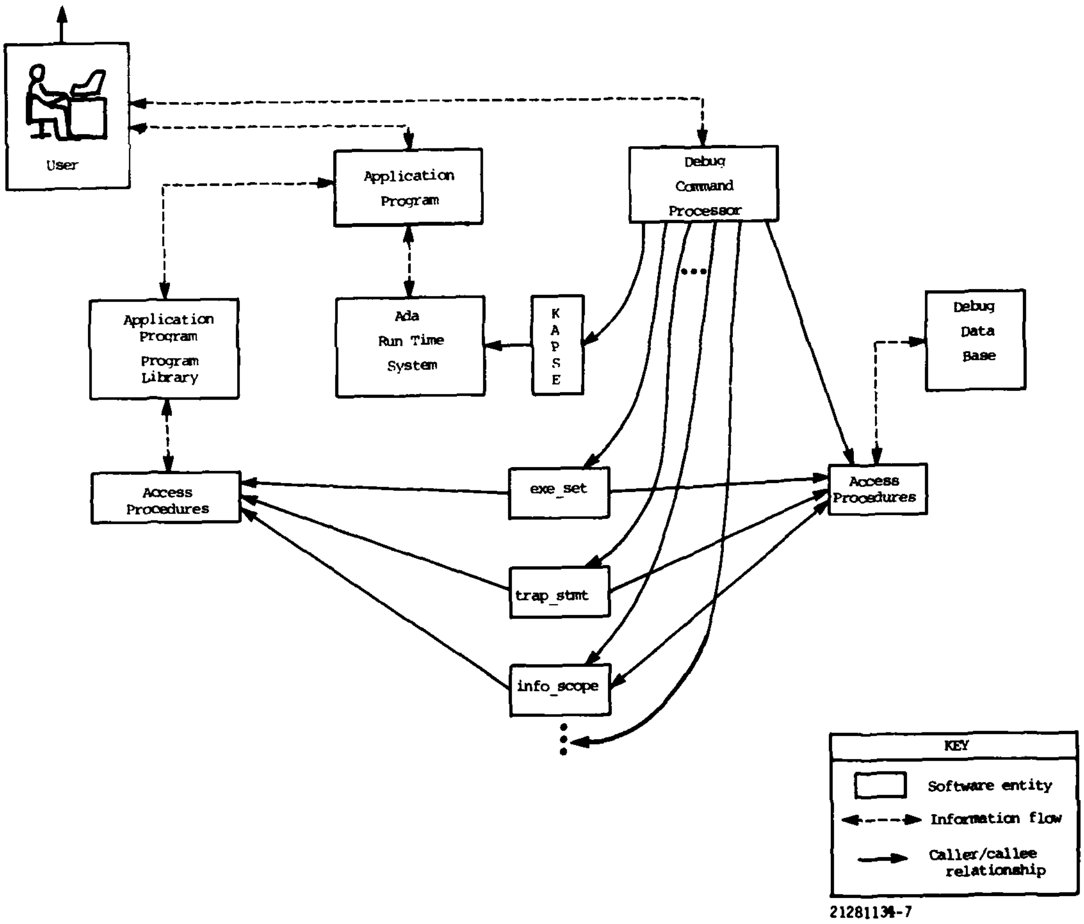


FIGURE 3-2: Debug Procedures and Interfaces

The OPTIMIZE parameter has a set of options that direct the Compiler to turn off specific sets of optimizations. The CAN_MODIFY and OFF values of the OPTIMIZE parameter provide the user with complete confidence that the Execution Control Commands have exactly the effect the Debug user expects. The CAN_INSPECT value of the OPTIMIZE parameter permits the Debug user to do only a DISPLAY of values in his program. The Execution Control Commands are not guaranteed to have their intended effect.

The ON value of the OPTIMIZE parameter permits the Compiler to move code and do other optimizations. The effect is that the debugger is not able to guarantee a reasonable result for any user command.

When a user issues a Debug command that might not have the intended effect due to the OPTIMIZE parameter of the specific compilation unit, the debugger prints a brief warning message and then processes the command.

The defaults of these compiler directives are DEBUG(ON) and OPTIMIZE(CAN_MODIFY). Thus, the usual mode of compilation permits the user to use Debug commands with full confidence that they will do as he expects.

3.2.4.3 Debug/RTS-KAPSE Interface

Debug control over user program execution is accomplished by directives to the Debugging Support Routine within the Ada Run Time System that is controlling the execution of the user program. The directives are divided into four distinct areas: execution control, value manipulation, exception handling control, and history trace.

The KAPSE is the interface between DEBUG and the RTS; that is, Debug calls the KAPSE whenever it wants to pass control to the user program, or whenever it wants to direct the RTS to perform some specific function. The KAPSE passes control and any information to the RTS. When the RTS encounters an active breakpoint in the user program, or has completed doing the work required, it calls the KAPSE, which returns control to the Debug.

Various pieces of information need to be passed between the RTS and Debug. The following sections describe that information, as well as defining what a hook is. Details of the implementation of hooks, the procedures that do the work, and the information passed across the KAPSE boundary are described informally.

In general, the information passed across the KAPSE is a set of codes representing specific actions to take or indicating events that have occurred, and an object representing a value on the target machine. This value is either the result of a memory fetch requested by Debug, the identity of the breakpoint encountered, or the identity of the exception encountered.

(a) Hooks. The actual control of the user program on a statement-by-statement basis is accomplished via hooks in the object module. Hooks are special branches to the Ada Run Time System. They are installed in the object module by the Ada Compiler under control of the DEBUG parameter. When the parameter is ON, hooks are installed between every statement of the compilation units affected by the parameter. For the purposes of debugging, declaration items are considered as statements, as are blocks, loops, and subprogram entry and exit.

Special care is taken in the Run Time System and in the Compiler code generator phases to assure that these calls to the Run Time System are as small and as efficient as possible. Each hook carries with it a unique identification number within the compilation unit. This identification number provides an index into a table, associated with the compilation unit, containing information about the hook, most importantly its memory address. Debug uses this memory address when directing the Run Time System to activate the hook. This table ıs contructed partially by the Compiler, and then is updated by the Linkage Editor with the actual memory address of the hook.

There are three distinct kinds of hooks, only two of which are generated by the compiler. These two are the normal statement hook, and the subprogram entry/exit hook. The compiler always puts hooks at procedure entry and exit. For this reason, the user can always perform procedure-level debugging. In addition, the RTS is always able to provide the user with a calling trace of his program, should it terminate abnormally. The normal statement hook is only installed when the compiler DEBUG parameter is ON. Otherwise they are indistinguishable. The third hook takes the place of either of the other two only when Debug tells the Run Time System to activate a specific breakpoint.

(b) Execution Control. The KAPSE provides a mechanism so that when a special character (interrupt) is typed on the user terminal, the RTS gets an interrupt that it recognizes. In the presence of Debug, the RTS transfers control to it through the KAPSE interface. When Debug is not active, the RTS suspends the execution of the program and returns control to its parent (normally the MCP).

The debugger can tell the RTS to start or resume program execution. This interface also includes a specification of a new address at which to resume execution. This is used for the Pgm_goto procedures.

Additional pieces of information are kept by the RTS so that it remembers its own state. Specifically it has the following states:

1. no debugger and, therefore, no breakpoints;

2. no breakpoints activated, but there is a debugger;

3. some specific breakpoints activated;

4. single step activated, return to debugger at every hook.

In addition, there are specific states for exception handling. This state information is used to control the RTS interface with the specific hooks in the user program. In this manner, more efficient execution is provided for non-debug scenarios.

In the case that some breakpoints are activated, Debug asks the RTS to modify the hook so that it behaves differently from a hook representing a statement with no breakpoint. Thus, control only passes to Debug when an activated breakpoint occurs in the user program. The two types of hook always transfer control to the RTS since the user can activate single step mode at any time. No modification of the user program is necessary to implement single step; only a directive to the RTS, telling it to return after every hook.

Future APSE extensions might provide for more information stored in the hook table. Alternative debuggers, pseudo-timers, or environmental simulators can use this mechanism with the extra information to provide full functional simulation of a target environment. See Section 3.3 below.

(c) Value Manipulation. The user can specify that Debug modify as well as print the values of program variables. Debug accomplishes this by calling the RTS (through the KAPSE), supplying an address and a value to be stored there or supplying a place to store the value retrieved from the program.

(d) Exception Handling. Debug can cause the RTS to trap specific exceptions, all exceptions, or only unhandled exceptions. The RTS raise handler does not unwind the stack until it finds a handler for the specific exception. (The run-time stack contains information from which the raise handler can compute which stack frames handle which exceptions.) This means that the user's program context is saved when an unhandled exception occurs rather than having the program unwind to top level. Control returns to Debug and the user can then take corrective action. This exception handling mechanism is always used, even when the Debug is not present at the time of the exception. This permits the user, once his program has terminated due to an unhandled exception, to invoke Debug, examine his program state, and take corrective action as necessary.

The mechanism for handling specific exceptions is as follows. Each TRAP EXCEPTION command issued by the user causes the debugger to look up the exception name and pass its identification to the RTS. The RTS keeps a list of exceptions it needs to check for whenever the RTS "raise" handler gets called.

(e) History Trace. The RTS maintains a history trace of the execution of the user program. This information is available to Debug. The information includes the full nested call chain of procedure and function calls. Also included is a limited history of the task scheduling and rendezvous activity as well as flow information. This information is kept in a table with a limited size, so that only the most recent events (with time stamps) are recorded. See the BACKTRACE command.

3.2.4.4 Debug/Internal Data Base Interface

The Debug internal data base is an abstract type implemented as a package. The data base keeps track of active breakpoints, the commands associated with a breakpoint, the current scope, as well as states of Debug. These states include: (1) at a breakpoint; (2) doing single step; (3) processing stored commands.

3.2.4.5 Debug/Program Library Access Package

Debug requires access to specific information stored in the Program Library for the executing program. The entire symbol table for each compilation unit that makes up the executing program is used by Debug to determine which variables the user is referring to in a DISPLAY, or TRAP MODIFY or SET command. This symbol table includes full storage allocation information for each variable, as well as an enumerated type indicating the optimization level and presence of hooks in each specific compilation unit. The addresses of variables are necessary for the DISPLAY and SET Debug commands. The information concerning optimization level and presence of hooks is used by the debugger to warn the user if some command may not have its intended effect.

A map of all the symbol tables of the executing program are necessary as well. This map is created by the Link Editor. Debug uses this to determine the new scope when the user gives a SCOPE ENCLOSING or SCOPE TO command.

A table of each hook location is also created by the compiler. Debug uses this table, in conjunction with the map produced by the link editor, when directing the Run Time System to activate or deactivate specific breakpoints.

The interface between Debug and the Program Library is a package specification. Following are the specifications for two specific procedures in that package.

(a) Analyze. This procedure takes a program environmment and an t abstract syntax form of a Diana tree for an Ada expression and produces a completed Diana tree. The program environment is a pointer into the Library Data Base specifying a specific point in the user program. The completed Diana tree is a post-semantic representation; it does not include code generation. This permits the debugger to access the variables in the user program's address space. The design of this procedure follows the design of the semantic analysis phase of the Compiler, and specifically uses the Lookup procedure.

(b) Cref_look. This procedure inspects the cross-reference data base constructed by the Compiler in the Library. It finds the correct variable and returns the list of points where the variable is used in assignment context. This procedure is used to implement the TRAP MODIFY command.

3.2.4.6 Breakpoint Command Procedures

A separate procedure exists for each option of the BREAKPOINT command since the work they do is sufficiently different. Each procedure takes three fixed arguments in addition to any it needs for itself. These are the user specified name of the breakpoint (if any), the list of breakpoint identifiers (if any), and an enumerated type giving the option specified by the user (establish, remove, suspend or restore breakpoint).

(a) Trap_stmt. This procedure takes the three standard breakpoint arguments. Additionally it takes a list of statement identifiers.

The Trap_stmt procedure does the processing necessary to set a breakpoint at the specified statement in the user program. It tells the RTS which statements have traps set for them and makes the necessary marks in the debugger database.

(b) Trap_label. This procedure takes the three standard breakpoint arguments. Additionally it takes a list of labels in the user program.

The Trap_label procedure does the processing necessary to set a breakpoint at the statement specified by each of the labels in the user program. The labels can be qualified like statement identifiers or unqualified. In the latter case, the label must be visible in the current scope.

(c) Trap_every. This procedure takes the three standard breakpoint arguments. Additionally it takes an integer.

The Trap_every procedure does the processing necessary to repeatedly cause a breakpoint to occur every "n" statements. The statement interval is specified by the integer argument. This integer gets saved in the Debug data base along with the breakpoint command. The RTS is told to enter single step mode, and given the integer. The RTS counts statement hooks, and returns to Debug when the specified number of statements have occurred. If hooks are turned off in some compilation units, the Trap_every procedure can only count subprogram entry and exit as statements, since no other hooks are present. This statement/hook count ignores scheduling of various tasks; that is, every n^{th} statement/hook, regardless of context, that is executed is counted.

(d) Trap-modify. This procedure takes the three standard breakpoint arguments. Additionally it takes a list of variables.

The Trap_modify procedure does the processing to set a breakpoint before every statement in which the specified variables could have their value changed. These statements are the assignment statements, and procedure calls that have the variable as an OUT or INOUT parameter. The procedure accesses the cross reference tables generated by the compiler to determine this information, and sets the necessary breakpoints by entering them in the data base and telling the RTS to activate them. A complete variable must be specified, except that components of record variables may be specified. Components of arrays are specifically not permitted.

(e) Trap_exceptions. This procedure takes the three standard breakpoint arguments. Additionally it takes a list of exception names.

The Trap_exceptions procedure does the processing to cause control to return to Debug for the specified exceptions. It does this by telling the RTS to trap the specific exceptions, and making any necessary marks in the data base.

(f) Trap_all. This procedure takes the three standard breakpoint arguments.

The Trap_all procedure does the processing necessary to cause control to return to Debug upon every exception in the user program. It does this by telling the RTS to trap all exceptions and making any necessary marks in the data base to record the state.

(g) Trap_unhandled. This procedure takes the three standard breakpoint arguments.

The Trap_unhandled procedure does the processing necessary to cause control to return to the debugger whenever an unhandled exception occurs in the user program. It does this by telling the RTS to only trap unhandled exceptions, and making necessary marks in the database to record the state.

(h) Trap_invoke. This procedure takes the three standard breakpoint arguments.

The Trap_call procedure does the processing necessary to cause control to return to Debug whenever a subprogram (task, entry, procedure and function) is entered. It does this by activating the RTS subprogram entry trap and making any necessary marks in the database to record the state.

(i) Trap_return. This procedure takes the three standard breakpoint arguments.

The Trap_return procedure does the processing necessary to cause control to return to Debug whenever a subprogram (task, entry, procedure and function) is exited. It does this by activating the RTS subprogram exit trap, and making the necessary marks in the database to record the state of the breakpoints.

(j) Trap_bkpt_id. This procedure takes a list of breakpoint identifiers and an enumerated type specifying what action to take.

The Trap_bkpt_id procedure does the processing necessary to remove, suspend or restore breakpoints and their associated actions. The action applies to the entire list of breakpoints specified by the user. Breakpoint identifiers can themselves refer to groups of breakpoints. Thus, the user can manipulate groups of breakpoints with single commands.

3.2.4.7 Execution Control Command Procedures

The following procedures do the processing necessary for the execution control commands.

(a) Exe_set. This procedure takes a single tree representing the assignment statement.

The Exe_set procedure does the processing necessary to cause the assignment to take place. First the Analyze procedure is called to correctly associate the identifiers in the tree with their semantically correct entities. The Evaluate procedure is then called to evaluate the expression, and the address of the variable. The results are then given to the RTS to cause the value to be put in the correct memory address in the user program.

(b) Exe_goto_stmt. This procedure takes a statement identifier.

The Exe_goto_stmt procedure does the processing necessary to cause the transfer of control to the specified statement in the user program. The statement identifier is first checked to be sure that the statement is in the currently active procedure. The statement identifier is looked up in the program location table provided by the compilerr and the new program location is passed to the RTS. No check is made to see if the transfer is legal (not into inactive blocks etc). The user is responsible for assuring that the transfer is meaningful.

(c) Exe_goto_label. This procedure takes a label identifier.

The Exe_goto_label procedure is responsible for causing the user program to resume control at the specified label. The procedure checks that the label is within the currently active procedure and then calls the RTS to set the new execution location. No check is made to assure that the transfer is legal (into inactive blocks, etc). The user is responsible for assuring that the transfer is meaningful.

(d) Exe_return. This procedure takes an optional tree representing an expression.

The Exe_return procedure does the processing necessary to cause the subprogram enclosing the currently active breakpoint to return. The expression is used as the return value in the case that the current subprogram is a function. The expression is Analyzed and then Evaluated. The result is passed to the RTS and the RTS is told to do a "return" from the current procedure.

(e) Exe_raise. This procedure takes an argument representing the exception to "raise".

The Exe_raise procedure does the processing necessary to raise the specified exception in the user program. The exception identifier is analyzed to assure it is a legal exception, and the exception is then passed to the RTS which is told to raise it.

(f) Exe_proceed. This procedure takes an optional integer argument.

The Exe_proceed procedure does the necessary processing to cause the normal execution of the user program to proceed. The argument is interpreted to mean the number of times the currently active breakpoint is to be executed before returning to Debug. The default value is zero, meaning return to Debug the next time this breakpoint occurs. This number is entered in the data base and associated with this breakpoint. It is decremented each time the breakpoint occurs and, when it reaches zero, control is passed to the user interface part of Debug or to the set of commands associated with the breakpoint.

(g) Exe_step. This procedure takes an integer argument.

The Exe_step procedure does the processing necessary to cause the user program to be executed one statement at a time. The integer is interpreted as the number of statements to proceed before returning to Debug. The count is entered in the data base, and Debug state is set to "single step". The RTS is told to return to Debug at every hook. When control is returned, the count is decremented, and control returns to the Debug/command/processor when the count returns to zero. If control returns to the Debug command processor because of an established breakpoint or interrupt, the count is reset to zero (the single step command is stopped).

3.2.4.8 Information Command Procedures

The following procedures do the processing necessary for the Information Commands.

(a) Info_display. This procedure takes a list of variable names and an optional integer as arguments.

The Info_display procedure does the processing to display the values of the specified variables in the specified base on the user terminal (or file). The variable names are "analyzed" to determine their correct definition in the current context. This context is affected by the SCOPE commands, so the "current context" is not necessarily the same as the context of the "currently active breakpoint". After analysis, the variables are passed to Evaluate and their values fetched from the user program through a call to the RTS.

The integer must be an integer in the range 2 through 16. If no integer is supplied by the user, the variables are displayed in their default mode (fixed, floating, integer, string, boolean, enumerated literal, etc.) Array and record variables are displayed in their entirety with identifying labels (subscripts, field names, etc.). The base applies to all the variables on the list.

(b) Info_backtrace. This procedure takes an enumerated literal as an argument.

The Info_backtrace procedure does the processing necessary to print history trace information on the user's terminal (or file). The enumerated literal represents the kind of backtrace information required. Each one requires a call to the RTS to get the specific information, which is then printed out. The RTS keeps the information itself. The options are:

CHAIN -- Show the nested calling chain from program startup. Included are the parameter names and values for the most recent subprogram.

FLOW -- Show the recent flow of control (goto, if, etc). This information is maintained by the RTS in a ring buffer. Thus only a limited amount of history is saved. The information will be compacted as the program runs so that the maximum amount of information can be saved. One such compaction is to fold sequential statements into a single information unit. This information is gathered by the RTS's catching control at each hook and retaining the hook's identifying number.

TASK -- Show the history of multi-tasking flow. Each time the RTS schedules a specific task to run, it makes a note in a ring buffer (different than for flow). The note contains a time stamp as well, to help the user in debugging his program. Each time an accept or entry call is made, these events also get recorded with a time stamp. The note in the ring buffer contains sufficient information for Debug to provide statement identification for the program.

ALL -- This option causes all the other backtrace options to be executed one by one. In addition, the CHAIN option is modified so that the full interface of every subprogram on the calling chain is printed out.

(c) Info_scope. This procedure takes an enumerated literal and an optional scope identifier.

The Info_scope procedure does the processing necessary to change the scope in which Debug finds the definition of variables and other named entities referred to by the user. The effect of the scope command is forgotten once the user program is allowed to proceed from a breakpoint. When a breakpoint is encountered, the scope is set to the scope where the breakpoint has occurred.

The enumerated values permit the user to access scopes that he could not name explicitly. The TO option allows the user to specify an arbitrary scope. The CALLER option causes the scope to return one level up on the dynamic calling chain, permitting the user to look at the state of the calling subprogram. The ENCLOSING option changes the context to the statically enclosing scope of the current context. The RESET option returns the context to the place where the breakpoint occurred.

(d) Info_what. This procedure takes an enumerated literal as an argument.

The Info_what procedure prints out information concerning the current breakpoints, breakpoint commands and scope that Debug is manipulating. The enumerated literal specifies which of the options the user has specified. The SCOPE option prints out the identification of the current scope. The TRAP (or BREAK) option prints out the list of all breakpoints, suspended or not, with breakpoint identifications. These identifiers may be used for the abbreviated NO, SUSPEND and RESTORE commands.

(e) Info_dump. This procedure takes an enumerated type as an argument.

The Info_dump procedure does the processing to display all the variables in a specific scope. Normally the command refers to the current scope (affected by the SCOPE command). If the user specifies the ALL option, all scopes that are "active" (contain valid data) are "dumped". This involves extensive use of the Program Library Access Package to get the list of variables locations and names in a specific scope, and then calls to the RTS to get the current value of these variables. The DUMP command includes subprogram parameters as variables in this processing. Each scope and identifier is identified in the printout.

3.2.4.9 Debug Control Command Procedures

(a) Dc_save. This procedure takes a file name as an argument.

The Dc_save procedure does the processing necessary to save all the breakpoint commands in the Debug data base in a format to be read in by the MCP. This allows a user to accumulate a specific set of breakpoint commands and then save them for future work. The information is no different than that printed out by the WHAT TRAP command, only it is guaranteed to be readable by the Debug Command Processor as normal commands.

The name syntax for breakpoint commands is used to assure that all breakpoints retain the same names and numbers as in the current session.

No attempt is made to preserve either the current scope name or the state of the user program's execution. Neither is there an attempt to save the state of the output command's file. The only information being saved is the set of breakpoints.

(b) Dc_brief. This procedure takes a Boolean argument.

The Dc_brief procedure does the processing to control the messages sent to the user terminal. If the Boolean argument is true, Debug uses shorter messages when communicating with the user. It only affects terminal printout; file printout remains verbose. If the argument is false, verbose mode is turned back on. The data base has a flag that contains the state of this mode.

(c) Dc_append. This procedure takes a file name and an enumerated literal as arguments.

The Dc_append procedure does the processing to cause a copy of breakpoint and display output to be directed to a file. In the case that the enumerated literal specifies ONLY, no output is directed to the user terminal, except when control returns to the command level. In that case, the user is always prompted with the breakpoint identifier. The processing includes opening the file for append access, and making marks in the data base referring to the file and preventing terminal output if requested.

(d) Dc_no_append. This procedure takes no arguments.

The dc_no_append procedure does the processing to terminate output to the file specified in a previous APPEND command. Output to the terminal is restored. This is done by making marks in the data base specifying the debugger status.

3.2.4.10 Utility Procedures

These utility procedures are used by several of the Debug command procedures.

(a) Parse. The Parse procedure is responsible for parsing the subset of legal Ada expressions and Ada name references that are supported by Debug. The output is an abstract syntax tree that can, by further analysis, be turned into a legal Diana tree.

(b) Evaluate. The Evaluate procedure is responsible for evaluating a Diana tree that represents an expression. It is used by the DISPLAY and SET commands to evaluate the expressions specified by the user.

3.3 Adaptation

3.3.1 Debug Size Restrictions

Debug has no size restrictions other than those imposed by the KAPSE. There are no parameters to alter to make it have a larger data base, etc.

3.3.2 Debug Extensions

Debug is carefully designed to permit extensions to the set of commands for future APSE development. It is expected that APSEs will use the debugging facilities as a test bed for debugging embedded software applications by writing various control scripts and possibly extending its set of commands and control over the user program execution. The nature of these extensions is expected to be in the direction of environmental and functional simulation of the target environment.

3.3.3 Run-Time System Parameters

There are two size restrictions on the functionality of the Run Time System. Both are size restrictions on the amount of information the Run Time System can keep during the program execution. The first is the size of the execution trace. Initially, the RTS can keep only the most recent 100 to 150 execution events for the BACKTRACE commands. Changing the size of the table that holds this trace will allow larger trace of execution to be kept by the RTS, and thus be printed out by Debug. There are two such execution event tables : one being the task-rendezvous table, the other being the jump-flow table, one for each task.

The second size restriction is the number of traps on exceptions that the RTS can support at one time. The specific exception identifiers must be kept by the Run Time System for the exception identifiers must be kept by the Run Time System for the sake of efficiency. Initially, the RTS can remember 100 separate exceptions that the user has specified traps for. This size can be altered to allow traps on more exceptions.

4.0 QUALITY ASSURANCE PROVISIONS

The provisions to assure a quality product include testing at various levels, as well as the use of structured programming methodology. The structured programming methodology has aready been employed in the design represented by this document. Unrelated pieces of functionality are implemented by different procedures, whereas similar functions of different procedures have been separated out into a single generalized procedure.

See the CPDP [TBD] for a detailed discussion of the methodology to be used while constructing the MAPSE. For Debug, the following tests will be performed.

Each procedure implementing the action of one Debug command will be unit tested. This involves constructing a specific environment around the procedure to be tested, and then invoking it with specific arguments. The actions that the procedure takes can then be observed and compared with the expected actions.

The Debugger Support Routine that is a part of the Ada Run Time System will also be unit tested. This unit testing will be done without using the KAPSE to transmit codes to it. Rather, the procedures will be called directly from the test driver, and the results observed and compared with expected results.

The Debug/KAPSE-RTS interface will be functionally tested. This means that the debugger will call the KAPSE to manipulate the Debugger Support Routine, and the effects observed. A unit test is not necessary, since KAPSE testing should be complete.

The Debug Command Processor will be functionally tested. Unit tests are not necessary since the MCP will already have been tested. Only the extensions to the MCP need testing.

This completes the testing requirements for Debug. The tests will be repeated on both MAPSE configurations, VM370 and Perkin-Elmer 8/32.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

END

DATE
FILMED

2-82

DTIC